eRDF: Live Discovery for the Web of Data

Christophe Guéret, Paul Groth, and Stefan Schlobach

Department of Artificial Intelligence, Vrije Universiteit Amsterdam, de Boelelaan 1081a, 1081HV Amsterdam, The Netherlands

Abstract. eRDF is an infrastructure for exploring the Web of Data through evolutionary querying. The main idea is to employ the well-known strength of evolutionary strategies to find good, though possibly approximate, answers quickly. This allows us discover relevant answers to a user’s information need in an anytime way. As the system is based on the idea of guessing and verifying solutions, it does not require complex joins, which implies that we can easily query distributed data-sets (in our case live SPARQL endpoints) thus data can be both local and distributed. This allows eRDF to scale, e.g., our current system provides access to the Billion Triple Challenge (BTC) data set plus several other large datasets. Another important feature of our methodology is that it is robust against complex SPARQL queries, which is a crucial feature for discovery queries.

The basic functionality of our infrastructure is provided by a simple exploratory SPARQL endpoint, which allows discovery queries over the BTC and several other datasources. To show the potential of the infrastructure we implemented a prototype web application, Like?, which allows users to discover resources across the Web of Data.

1 Problem Description

The Web of Data is growing at an amazing rate as more and more data-sources are being made available online in RDF, and linked. At the same time specialised triple-stores, such as Virtuoso[9], OWLIM[1] or 4store[6], have matured into powerful engines that can efficiently answer queries for a given schema over static data sets of billions of RDF triples.

However, in many cases the schema is not known, nor is the precise nature of the search query. As the name suggests, query engines are suitable for precise querying, but necessarily fail when the task is more explorative, when the user needs to discover information, first. A second drawback of current approaches is that static data sets are explored and queried rather than the actual data sources themselves. It is acknowledged that the currently most convenient use of Semantic Data is by querying collections of static data, which are often outdated, instead of live discovery. This is due to the difficulty of joining results from different engines in federated querying. Finally, given the open character of the web, which is intrinsically incoherent, incomplete and incorrect, an exploration engine for the Web of Data must be robust. We claim that the eRDF infrastructure makes significant steps in these four areas: exploration, live-access,
decentralisation and robustness. We now discuss these areas in more detail before discussing eRDF and its use in the Billion Triple Challenge (BTC).

**Discovery queries** The paradigm shift on the WWW from browsing to search was one of the critical elements for its success as it allowed users to find relevant information without knowing its exact location in the network. In search users define their needs by providing keywords often with the goal to find relevant information without having a specific information source in mind. While semantic search engines, such as sig.ma, are beginning to provide search over the Web of Data, there is still the need for new techniques to discover what data is available, particularly, for software agents. Indeed, generating queries for a given data source usually requires extensive knowledge of that data-source in order to produce reasonable results. By integrating an approximate component into the query process, eRDF can aid discovery.

**Anytime answers over live distributed data-sources** Many of the applications based on the Web of Data do not use data sources directly, as federated queries over live SPARQL endpoints is known to be extremely expensive, because known optimizations (for example to deal with joins) do not work in the distributed case. Instead, snapshots are taken at intervals, dumped into gigantic repositories and made available in database style for querying. The effect is that the available information is constantly outdated, not just the index (as in traditional search engines), but even the data itself.

eRDF allows distributed queries over live data-sources as only very simple unary queries are needed. Additionally, eRDF can issues all of its queries in a fully parallel fashion. There is no theoretical restriction on the number of data-sources and their data-size only marginally increases individual response time. Of course, increasing data-size in combination with a constant population size will increase convergence time. However, given the any-time character of evolutionary methods good answers are still returned comparatively quickly. This makes eRDF an interesting alternative for exploration and discovery for the Web of Data.

**Robustness** Although SPARQL has been developed as an RDF query language for Web data, there is a discrepancy between the database like query formalism and the adaptive, open-world, incoherent and inconsistent character of the Web of Data. Schemas are often unknown, and posing promising queries requires explicit knowledge of the structure of the information. The effect of this is that many good answers are missed as queries are simply not adequate for certain information needs. eRDF does not extend SPARQL but releases some Semantic constraints if required by the application. This makes it more robust for querying unknown information, which is essential for exploration and discovery.

## 2 The eRDF infrastructure at a glance

In [8,7] we introduced RDF query answering by evolutionary algorithms (eRDF). The basic idea is simple: instead of indexing the triples and joining results of ground queries, we guess a population of candidate solutions. Those are then
improved by the classical mutation operation guided by a fitness function which, roughly said, calculates a distance of a candidate from being a solution. This distance can simply be the number of invalid triples in our solution, or more complex combinations of such simple metrics with user-defined similarity measures. Based on such well-defined, and user-specified, notions of similarity eRDF returns “perfect” answers if possible, and approximate answers if necessary, which is exactly what is required for discovery queries.

The input to eRDF is a standard SPARQL query. Currently, we limit our system on answering select queries making use of one or more WHERE clauses of simple graph patterns. As the time of writing, simple filter expressions assessing the equivalence of terms are also being implemented.

Let us consider a query with its set $G$ of graph patterns, its set $F$ of filter constraints and its set $V = \{?v_1, \ldots , ?v_n\}$ of variables to instanciate. A solution to that query is a mapping $\mu : V \mapsto I \times L \times B$ associating to every variable of $G$ a URI, blank node or literal taken respectively from the sets $I$, $B$ and $L$ of all the URIs, Bnodes and Literals present in at list one of the endpoint. Note that only an abstract representation of those sets is actually provided by the data layer, they are not actually created.

We will later need the notion of a candidate solution to a SPARQL query which is simply any mapping $\mu$ which assigns to every variable a node from one of the graphs.

**Evolutionary algorithm** An evolutionary algorithm is a population based heuristic. A set of candidate solution is improved in a generational process. Our proposed method makes use the fact that we can rank approximate solutions according to their similarity wrt. a perfect answer, in order to pick the candidates that we consider as off-spring for new generations. First, let us describe the general evolutionary algorithm.

The evolutionary algorithm presented in this paper considers a set of candidate solutions $P = \{\mu_i\}, i \in [1, |P|]$ as its “Population”. During the iterative optimisation process, the content of the population is improved by replacing all but one candidate solutions (the “individuals” of the population) by better ones. Evolutionary loops usually consists of the following steps: create an initial population, generate a new generation, and select the best solutions to be the new generation and loop to the generation of new individuals [5]. Within this loop, several operators may be used to obtain different behaviours of the evolutionary process. Our algorithm uses a (1,10)-ES evolutionary strategy[5] meaning that at every generation 10 candidates solutions are produced and only the best one survives. The generation of new individuals is driven by a local search heuristic: every new candidate solution is a slightly altered version of the best solution found in the previous generation.

---

1 Arguably, different user needs, and different data sets require different notions of approximation, and our evolutionary querying paradigm of eRDF is particularly suitable for integrating such different notions by combining user specified similarity measures within one querying paradigm.
Fig. 1 gives an overview over our general loop: from a user input we issue a discovery SPARQL query to eRDF for which expect to get approximate answers. Hereafter, we describe the individual steps in more detail.

**Initialisation** The population is initialised with some candidate solutions. Traditionally within the EA community, the candidate solutions are random solutions created from the search space. We have chosen instead to initialise the population to default solutions where all the variable are not bound this results in less queries being issued.

**Validation** The validation step consists in the evaluation of the candidate solutions. For every one of them, a quality score $\text{fitness}(\mu)$ is computed based on the quality of the bindings $\mu$ contains.

$$\text{fitness}(\mu) = \frac{1}{|V|} \sum_{?v_i \in V} \text{reward}(?v_i)$$

Any kind of rewarding scheme can be used under the only constraint that $\text{reward}(?v) \in [0, 1]$. A value of 0 denotes a bad assignment, a value of 1 a perfect one. Any intermediate value denotes a binding that partially fulfills the objective.

For instance, let us consider a rewarding scheme based on the quality of the query graph instantiated with the candidate solution: for each triple a score
is calculated depending on whether it exists (at least partially) in one of the data-stores or not. This reward scheme is currently used in our prototype.

For a graph pattern \( g = \langle ?v_1, p, ?v_2 \rangle \) and the candidate solution \( \mu : \{ ?v_1 = \text{foo}, ?v_2 = \text{bar} \} \). The reward given to \( ?v_1 \) and \( ?v_2 \) will be maximum if the instanciation of the graph pattern is a valid triple. This reward will be 0 if that generated triple does not exists. The reward for \( ?v_1 \) is computed as:

\[
\text{reward}_g(?v_1) = \begin{cases} 
1 & \text{if } \text{ask}(\langle \text{foo}, p, \text{bar} \rangle) = \top \\
0.5 & \text{if } \text{ask}(\langle \text{foo}, p, ?\text{whatever} \rangle) = \top \\
0 & \text{otherwise} 
\end{cases}
\]

the reward received for every graph pattern \( \text{reward}_g(?v) \) are then averaged to obtain a global reward \( \text{reward}(?v) \).

In these two equations, the \( \text{ask} \) denotes a standard ASK query expressed in SPARQL. That query is sent to the datalayer which in turn send it to the different SPARQL endpoints it is connected to. The variable “?whatever” is used as a wildcard to test the partial validity of a triple.

As these \( \text{ask} \) queries are simple validation queries on a single graph pattern, this step is very efficient. We also make use of a cache to further optimise this verification.

\section*{Selection} The fitness value of the candidate solutions is used to rank them. Considering two candidate solutions \( \mu \) and \( \mu' \), \( \mu \) is better than \( \mu' \) if \( \mu' \text{ fitness}(\mu) > \text{fitness}(\mu') \). According to our (1,10)-ES selection strategy, all the candidates solutions are sorted and the population is reduced to just one element; ie. only the best individual survives. That best individual is also copied to a result bucket, waiting there to be fetched by the client. This buffering strategy allows us to always return the best-so-far solution to the client.

A candidate solution that managed to stay the best one of the population for 5 consecutives generations is assumed to be (locally) optimal. When encountered, such a solution is stored in a tabbu list and the search continues in another part of the search space.

\section*{Create offspring} In this step, the best candidate solution found on the previous step is modified with the hope of improving it. This candidate solution is modified (mutated) 10 times to create 10 new candidate solutions. In order to create a new candidate solutions, some of the variables are given a new value. For instance, \( \mu' : \{ ?v_1 = \text{something}, ?v_2 = \text{bar} \} \) and \( \mu'' : \{ ?v_1 = \text{dummy}, ?v_2 = \text{bar} \} \) could be created by mutation of \( \mu : \{ ?v_1 = \text{foo}, ?v_2 = \text{bar} \} \).

The choice of keeping the value associated to a variable or changing it depends on the reward that assignment received in the validation phase. The lower that reward was, the higher are the chances for that variable to be mutated. The mutation itself consists of the assignment of a new value to a variable. For instance, \( ?v_1 = \text{foo} \rightarrow ?v_1 = \text{dummy} \). That new value is picked up at random from one of the data-stores.
3 Implementation and Infrastructure Setup

In the following, we detail the implementation of eRDF and the infrastructure setup for the BTC. The core of eRDF is implemented in Java 1.6. We built it upon well known frameworks and toolkits. Jena ARQ[2] is used to parse SPARQL queries, the evolutionary loop is based on ECJ[4] and a RESTful query interface is powered by RESTlet[10]. The source code of eRDF is publicly available under a GPL license.

eRDF along with the web application detailed below are run on a machine with a 2.8Ghz Dual-Core AMD Opteron(tm) Processor, 512M of RAM and 200GB of storage. eRDF requires roughly 40Mb of RAM to run over the BTC data set. In addition to the BTC data set, we configured eRDF to run over the following publicly available SPARQL endpoints:

<table>
<thead>
<tr>
<th>Source</th>
<th>Endpoint</th>
<th>Number of triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>US Census data</td>
<td><a href="http://www.rdfabout.com/sparql/">http://www.rdfabout.com/sparql/</a></td>
<td>1000 million</td>
</tr>
<tr>
<td>DBPedia</td>
<td><a href="http://dbpedia.org/sparql/">http://dbpedia.org/sparql/</a></td>
<td>274 million</td>
</tr>
<tr>
<td>MusicBrainz</td>
<td><a href="http://dbtune.org/musicbrainz/sparql/">http://dbtune.org/musicbrainz/sparql/</a></td>
<td>36 million</td>
</tr>
<tr>
<td>DBLP</td>
<td><a href="http://dblp.l3s.de/d2r/sparql/">http://dblp.l3s.de/d2r/sparql/</a></td>
<td>10 million</td>
</tr>
<tr>
<td>WordNet</td>
<td><a href="http://wordnet.rkbexplorer.com/sparql/">http://wordnet.rkbexplorer.com/sparql/</a></td>
<td>2 million</td>
</tr>
<tr>
<td>Revyu</td>
<td><a href="http://revyu.com/sparql/">http://revyu.com/sparql/</a></td>
<td>unknown</td>
</tr>
<tr>
<td>CIA World Factbook</td>
<td><a href="http://www4.wiwiss.fu-berlin.de/factbook/sparql/">http://www4.wiwiss.fu-berlin.de/factbook/sparql/</a></td>
<td>unknown</td>
</tr>
</tbody>
</table>

While the BTC data set includes some parts of this data set this is not know to eRDF, therefore, eRDF operates over well-above 2 billion triples. We now discuss our setup for hosting the BTC dataset.

For the BTC, we used a high performance server with 8 processors, 32 GB of RAM and a 4.6 TB of storage. Each processor was 2.4 Ghz Quad-Core AMD Opteron(tm) Processor. For triple storage, we selected the quad-store, 4store[6]. We ran 58 instances of 4store on a single server where each instance exposed roughly 20 million triples. There were two reasons to run this number of instances: to test the distributed query capabilities of eRDF; and to ameliorate the performance degradation that 4store experiences with large numbers of properties, which is one of the characteristics of the BTC dataset.

4 Use Case and Discovery Frontend

To demonstrate the use of eRDF over the BTC dataset, we focused on a discovery use case, namely, the ability to find things that are similar to or “like” a given entity. We term such a query a like-search. For example, a user that is new to a city may want to find people like themselves. Alternatively, a user may want to find a holiday destination that is similar to the one they travelled to last year. Furthermore, such queries may be useful for finding unrealized connections between entities. Take for instance, a social scientists who wants to characterize a particular university. Often this is done by grouping universities together on various dimensions, for example, student population size, number of faculty, etc. However, by first performing a like-search, the scientist may find dimensions
that she or he may not have considered, for example, size of sports teams. These are just some examples of like-searches.

Fig. 2. Finding entities similar to Tim Berners Lee

In order to demonstrate like-searches, we built a web application, Like?, over the top of eRDF. The interface to the application is shown in Figure 2. It shows the results of a like-search on Tim Berners Lee. To the left of the screen, the user can enter full text queries. These queries are forwarded by Like? to the semantic search engine Sindice [3]. From the response, the first result is selected and the RDF-document it refers to is retrieved. This document is then transformed into a SPARQL query, which is issued to eRDF. The SPARQL query produced in the case of the Tim Berners Lee query contains 23 graph patterns. The document selected for the like-search is displayed to the user at the top of the page, in this case, the DBpedia page describing Tim Berners Lee. Answers are displayed in the right half of the screen along with the number of triples, shown in parenthesis, that matched the given query. Users can hover over each answer to see the property and object of the triple. Finally, Like? will continue to return possible answers to the query until the user clicks stop or a server-side timeout is reached.

We note that the answers given by eRDF are both expected and novel.

---

2 Isn’t Tim Berners Lee the David Beckham of computer science?
example, is Wendy Hall, who like Tim Berners Lee, is a well known British computer scientist who also holds a professorship at the University of Southampton and is a fellow of the Royal Academy of Engineering. Some more novel examples are Ada Lovelace and Thomas Malthus both of whom are English scientists as well Danny O’Brien an English technology journalist who blogs.

While Like? is in its early prototype stages, we believe it successfully demonstrates the use of eRDF for a discovery over the Web of Data. Like? is accessible at http://ai01.cs.vu.nl/erdfwww/likeengine/.

5 Conclusion

As the Web of Data continues to grow, it is becoming increasingly necessary to take a distributed approach to using it: acquiring data from sources on-demand in a robust fashion. Additionally, because of its size, new techniques need to be developed to discover the information that is there. The eRDF infrastructure provides a novel evolutionary technique to enable discovery over distributed SPARQL endpoints in a robust fashion. In this paper, we described eRDF and its usage for Like?, an application for finding similar things. This application runs over live data sources as well as the BTC data set.

References